

DSCI 551 Spring 2024

“BDK Distributed Database for USC Public Safety”

Final Report

by Dan Lee, Rodney Meeler, Kyle Wayne Parker - Team 6

<https://github.com/ShirPanjshir/551-database-project>

Introduction

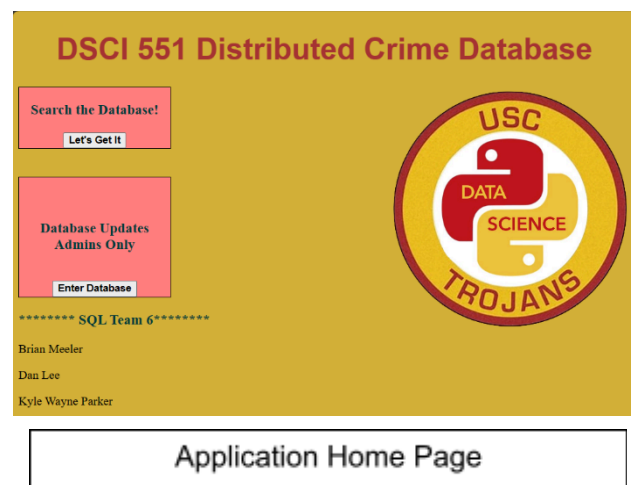
This distributed database application provides the user with a way to intuitively interact with USC Department of Public Safety information. It facilitates all standard database CRUD functions, as well as an Excel/CSV download option to personalize the interaction for each user. It is intended to employ all of the concepts learned over the semester and to be informative and to give back to the greater USC community that made this project possible.

Planned Implementation

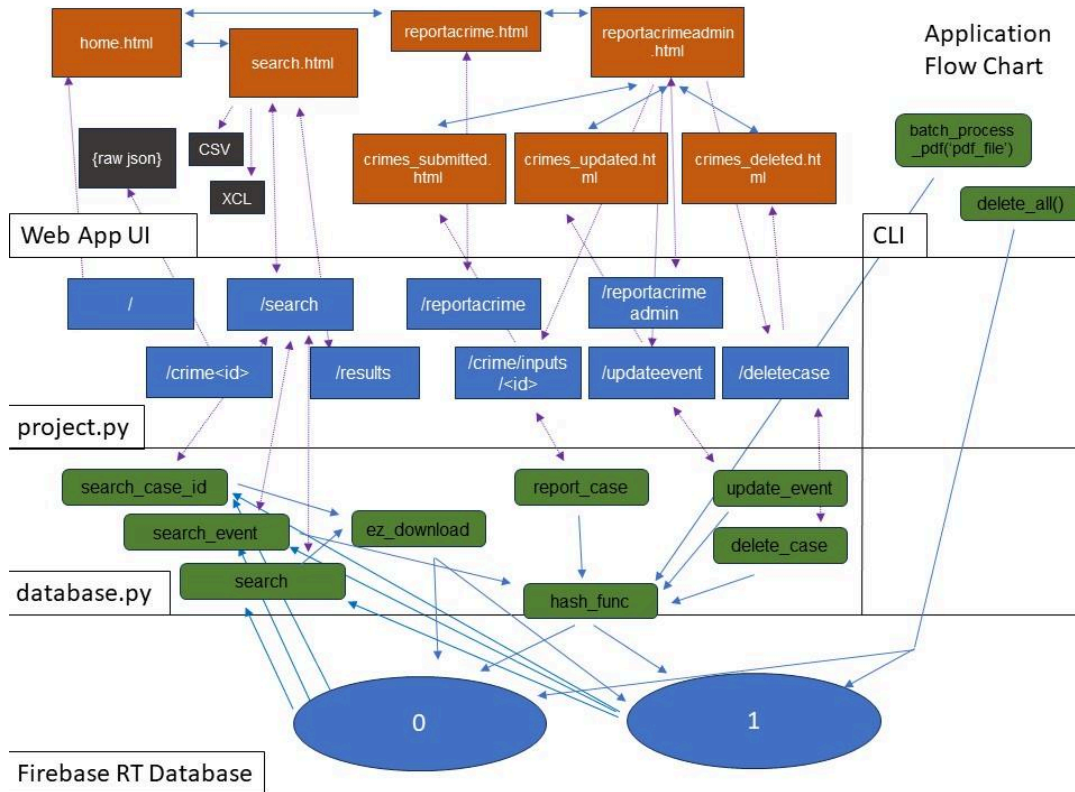
We will create a distributed database that stores crimes that have been reported to USC Public Safety. The dataset is structured with 10 columns per row and could easily be uploaded into a SQL database. However, a NoSQL database would make the system more flexible, especially for scaling out additional detail in subsequent columns. The distribution of databases should be initially determined by the case number: one for those with a case number and one for those without. Incidents with a case number are more likely to be accessed/modified for further updates. Incidents without a case number are usually less urgent or impossible to investigate and therefore this aspect naturally lends itself to separation.

Design and Layout

This distributed database application was built in Python Flask and HTML and connects to two Google Firebase Realtime Databases. The web interface is based around the home page, which directs users to “Search the Database” or to “Database Updates Admins Only” for creation, updates, and deletions. Author information is also



provided at the bottom. At any point users can return to the home page to begin another action. The data itself is derived from the USC Department of Public Safety Daily Crime and Fire Logs, which are PDFs and publicly available. The original 10 columns of the Logs are expanded by splitting “Offense”, “Initial Incident”, “Final Incident” into -Category and -Description, e.g. “Offense” became “Offense Category” and “Offense Description”. Similarly, “Location” was expanded into “Location” and “Location Type”. This provides 14 different ways to interact with and filter data to create a thoroughly detailed NoSQL record for every crime event. Partial and successive searches are possible, that is the database will match on partial entries and multiple fields can be searched against simultaneously to narrow down to even a single report. The application is light on styling and format to ensure maximum efficiency and ensure the user experience is fast and accurate. The accompanying Powerpoint, entitled “TEAM 6 - The BDK Distributed Database for USC Public Safety” provides a thorough graphical design review of each page used in the application. The following flow diagram visually depicts the layers of networking and overall functionality of the application.



The search page is split into three components: simple search across the top, advanced search underneath, and results on the bottom. The forms for “Search by Case” and “Search by Event Number” are in the top simple search section, as these should be the most frequently used tools of this database search tool for a Public Safety Official. The background for this section is pink, to visually group and separate itself from the advanced search area.

HOME

Search By Case Number

Search By Event Number

Enter CaseID Number

Search for Case

NN-NN-NN-NNNNNN

Search for Event

Advanced Search

Date_Reported YY-MM-DD

Date_From YY-MM-DD

Date_To YY-MM-DD

Disposition

Final_Incident_Category STRING

Final_Incident_Description STRING

Initial_Incident_Category STRING

Initial_Incident_Description STRING

Location STRING

Location_Type

Offense_Category

Offense_Description STRING

Advanced Search

Matching Database Entries: 119

Event	CaseID	Date_From	Date_Reported	Date_To	Disposition	Final_Incident_Category	Final Incide
23-12-06-1348506	2306592	2023-12-06 12:30	2023-12-06 12:30	2023-12-06 12:30	Open	SERVICE	Incident Report

Search Page – Showing Results of Query

The advanced search area, over a gold background, has fields for each of the remaining 12 fields in the database. Disposition, Location Type, and Offense Category are provided a dropdown window to select from a premade list of entries, created from summary analysis of the database. This dropdown saves time for the user and allows for rapid groupings.

The results area initially reads out “No Results Found” to indicate to the user that no search has yet been conducted. When a search term is entered into any of the three search functions: by case number, by event number, or advanced, the matching results populate on the same search page. The message will switch to “Matching Database Entries:” followed by a count of the entries. The table of matching entries is then populated below for the user to review. Displaying returns on the same page minimizes the number of HTML pages required and, if the desired results are not found, allows the user to review the data while entering in updated search terms. This secondary search window allows users to perform a fine-tune query of their initial results, without having to reassess the database. Here, a few lines of javascript allow for real-time sorting, sub-querying, and download of the remaining reports that are displayed. That is, if there are initially 100 reports returned, and a sub-query is performed via the small search bar above the results that reduces the number of matching reports to 50, only those 50 will be populated into a file if the download Excel/CSV button is selected. This ensures that

the user gets exactly what they see on the screen in their download, not the contents of the initial search, which is the most intuitive way for this operation to perform. Pertaining to the act of searching, each entry field utilizes a placeholder that shows the user exactly how to enter data into the field. This requirement of highly structured data going into the search field allows for a more efficient search function that employs minimal cleaning. The user demographic would be a trained employee, so familiarity with the required formatting is expected to be developed over time. If improperly formatted data is presented at any point, in any field, the application refreshes the page with a warning message to properly format data.

Manipulation of the database is done by clicking the “Enter the Database” button in the second table in the home page. This brings the user to an intermediate page that requests a password. This gateway acts as a simple interruption for unintended users. Upon entering the password “PLEASE”, the user is directed to the database interaction page. Failure to do so redirects to a dead-end that has a HOME for the user to return. As security was not the objective of this project, this page nevertheless provides a good placeholder for more sophisticated security updates to be made in the future, if so desired.

Similar to the search page, the manipulation page has three distinct blocks, appearing in order of anticipated probability of use: Report a Crime (CREATE), Update a Crime (UPDATE), and Delete and Event (DELETE). These sections all have unique colors, with delete specifically over a red field, as deletions of database records are generally not popular. To that end, any submission of a deletion will be met with a pop-up that requests users to confirm their deletion request. This small detail prevents accidental deletions and delays the process, both of which reinforce the preference of updates over deletions in databases. Reporting a Crime has all 14 fields represented similar to

HOME

Make Database Adjustments Below: Create, Update, Delete

REPORT A CRIME

Assign Case Number? ☐ Yes ☐ No

Date From

Date To

Disposition

Final Incident Category

Final Incident Description

Initial Incident Category

Initial Incident Description

Location

Location Type

Officer Category

Officer Description

Submit

UPDATE A CRIME

Event

CaseID

Date From

Date To

Disposition

Final Incident Category

Final Incident Description

Initial Incident Category

Initial Incident Description

Location

Location Type

Officer Category

Officer Description

Submit

DELETE AN EVENT

Event

Submit

Admin Page for DB Create, Update, Delete

the search page, with a bubble selection for assigning a case number, defaulted to Yes. Upon submission, the user is directed to a new page that displays an entry confirmation message with the location and newly generated event number displayed. Behind the scenes, a hashing function, based on the time that the record is created, is used to sort entries into either database. With this, users can copy the number and rapidly search for it, with minimal navigation. Two buttons, HOME and Return to Database are available. Again, improper entry of data into any field will refresh the page with a warning at the top to ensure proper formats are used and a confirmation that the data was not submitted. The update function is similar to the entry, with the caveat that an event number must be entered to specifically select which single record will be updated by entries in the fields below. All, some, or none of the fields can be adjusted per submission of the Update function. The delete function only accepts event numbers to specify and limit deletions.

Implementation

After cloning the repository, the user can interact with the application by running the project.py, which hosts the site on the user's 127.0.0.1:5000 socket, the default for Flask.

Additionally, an AWS EC2 instance with an elastic (static) IP address to host the application at <http://3.216.130.143> has been constructed. We had to enable port 80 (the default HTTP port) on the AWS website and then create an Nginx configuration file to make the local Flask server publicly accessible. We also set up a systemd file to keep gunicorn running the server even after ending the SSH session to the EC2 instance.

```
GNU nano 4.8 flaskapp
server{
    listen 80;
    server_name 3.216.130.143;
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
}
```

Nginx Configuration

```
GNU nano 4.8 flaskapp.service
[Unit]
Description=Gunicorn instance to serve myapp
After=network.target

[Service]
User=ubuntu
Group=ubuntu
# Environment="PATH=/usr/bin"
WorkingDirectory=/home/ubuntu/551-database-project-main
ExecStart=/usr/bin/gunicorn project:app

[Install]
```

Systemd service

Backend

The `database.py` houses nine functions that drive the application. All but two functions, `batch_process_pdf(pdf_path)` and `delete_all()`, are used on the web interface. The batch process function is not intended for daily use, but rather to seed the database with several thousand reports. As such, it is reserved as a command line interface function for the database administrator to utilize as new reports, the data sources, are published. This function utilizes the `tabula` package to parse the provided `.pdf` file and cleans and structures each row into JSONs that are then iteratively distributed to the database based on the hash function. The `delete_all()` function is impractical in the utility of the application, but useful for preparing the database for demonstration. It requires a verification code “8675309” for the function to activate, preventing any accidental execution.

`hash_func(event)`

Upon closer examination of the data, we concluded sending cases with a case number to one database and those without to the other would be impractical. Only 30% of cases we collected have a case ID, making even distribution of data now and in the future almost impossible. This hashing method also makes enforcing rules while adding/removing case ID very challenging. We decided to use the event ID, which is a unique number based on the time reported, as the hash key.

This function simply returns (last digit modulus 2) of the event ID. The last six digits of an event ID (YY-MM-DD-XXXXXX) are determined by the time the entry is created, down to the second. This hashing rule ensures an even 50% probability split for inputs.

This function is typically used in combination with the URL tuple (`db0_url`, `db1_url`) to locate the correct database. Future expansion can be easily done by adding more urls to the URL tuple and changing the division factor to match the number of databases.

Error Handling

All of the following functions were designed to be error-proof to minimize front-end development cost and streamline error handling. Instead of raising different types of Python errors, we accounted for all the foreseeable errors and developed a standardized and highly modular format to return the result:

- tuple(json data or relevant information, 200) if successful
- tuple(customized error abbreviation, error code) if the function encounters errors

We also designed our customized error abbreviations to account for all foreseeable errors and created the MESSAGE dictionary to allow quick retrievals of corresponding messages. This allows abovementioned Flask apps to easily check the status of the request using the second return value (200 or error code) and then render the result or error message accordingly.

```
MESSAGE = {'IDERROR': 'Incorrect input format - Case ID must be a seven-digit number.',
           'IDCONFLICT': 'This Case ID has already been used.',
           'CASEID_NULL': 'Please enter a CaseID.',
           'EVENT': 'Incorrect input format - Event Number must follow this format: NN-NN-NN-NNNNNN.',
           'EVENT_NULL': 'Please enter an Event Number.',
           'ADV_EMPTY': 'Please fill in at least one field.',
           'DTERROR': 'Incorrect date format or invalid date.',
           'DTCONFLICT': 'Date From must be earlier than Date To.',
           'WARNING': 'Incorrect input format - please adjust',
           'CONNECTIONERROR': 'Connection error - Please contact admins.',
           'NOMATCH': 'Event Number %s not in Database, no entries %s.',
           'SUCCESS': 'Case with Event Number %s %s.'}
```

ez_download(p=None)

This is a helper function that uses the same search parameters “p” to retrieve data from all databases using requests.get(). It returns ({combined json data}, 200) when all requests are successful and ("CONNECTIONERROR", 7700) if otherwise.

search_case_id(case_id)

The case id is a seven-digit number reserved for high priority cases. This search function first uses re.match to check if case_id has the correct format, then it passes case_id parameters into ez_download to retrieve the case. Possible return values:

- ({case}, 200) if successful
- ("IDERROR", 7700) the entered case ID is not a seven-digit number
- ("CASEID_NULL", 7700) no case ID was entered (empty string)
- ("CONNECTIONERROR", 7700) passed from ez_download

search_event(event)

Similar to search_case_id, this function uses re.match() to enforce input format (YY-MM-DD-XXXXXX). Since the data is partitioned based on event IDs, this function utilizes the hash function to narrow down the search to just one database. Possible return values:

- ({case}, 200) if successful
- ("EVENT", 7700) incorrect event ID format
- ("EVENT_NULL", 7700) no event ID was entered (empty string)
- ("CONNECTIONERROR", 7700) any connection error / r.status code != 200

search(start_dt=None, end_dt=None, date_rep=None, off_cat=None, off_des=None, ii_cat=None, ii_des=None, fi_cat=None, fi_des=None, loc_type=None, loc=None, disp=None)

The function used by the "Advanced Search" described above. It queries all databases to identify matches, including partial and case insensitive, through each provided argument field. All three date fields (start_dt, end_dt, date_reported) must follow the same datetime format: YYYY-MM-DD HH:MM, and end_dt cannot be earlier than start_dt.

Due to Firebase's limitations, it picks one of the categorical (offense_category, location_type, etc.) or time variables (if at least one of them was entered) for the ez_download function to minimize download size and then process the remaining filters locally through pandas. Possible return values:

- (JSON, 200) if successful. The JSON object contains all crime matches, with each key (event ID) value (columns) pair representing a case.

- ("DTERROR", 7700) incorrect start_dt/end_dt format
- ("DTCONFLICT", 7700) end_dt earlier than start_dt
- ("CONNECTIONERROR", 7700) any connection error / r.status code != 200

**report_case(caseid=False, start_dt=None, end_dt=None,
off_cat=None, off_des=None, ii_cat=None, ii_des=None,
fi_cat=None, fi_des=None, loc_type=None, loc=None, disp=None)**

This function is the main create function available to the user on the site. It uses pandas.Timestamp to generate time reported and event ID (Pacific Time). If caseid=True, it assigns a case ID by retrieving the latest case ID through ez_download and then adding it by one to ensure all case IDs are unique and sorted. It also verifies if both start_dt and end_dt are valid dates in YYYY-MM-DD HH:MM format and end_dt is not earlier than start_dt.

The function then combines all data into a JSON object and submits it to the database with the help from hash_func and requests.patch. Possible return values:

- (assigned event number, 200) successfully uploaded
- ("DTERROR", 7700) incorrect start_dt/end_dt format
- ("DTCONFLICT", 7700) end_dt earlier than start_dt
- ("CONNECTIONERROR", 7700) any connection error / r.status code != 200

**update_event(event, caseid=None, start_dt=None, end_dt=None, off_cat=None,
off_des=None, ii_cat=None, ii_des=None, fi_cat=None, fi_des=None,
loc_type=None, loc=None, disp=None)**

This UPSERT function first checks if the event exists through hash_func and search_event. If a case ID was entered, it also checks if the ID is available through search_case_id. It then checks date formats and analyzes both original and new date variables (if they exist) to prevent date conflicts in the final result. Possible return values:

- ("SUCCESS", 200) successfully updated
- ("NOMATCH", 7700) no event found
- ("IDCONFLICT", 7700) case ID has already been used
- ("DTERROR", 7700) incorrect start_dt/end_dt format
- ("DTCONFLICT", 7700) end_dt earlier than start_dt in the final result
- ("CONNECTIONERROR", 7700) any connection error / non-200 status code

delete_case(event)

This function deletes the event if it exists. Possible return values:

- ("SUCCESS", 200) event successfully deleted
- ("EVENT", 7700) incorrect event ID format
- ("EVENT_NULL", 7700) no event ID was entered (empty string)
- ("CONNECTIONERROR", 7700) any connection error / r.status code != 200

Flask-based Website

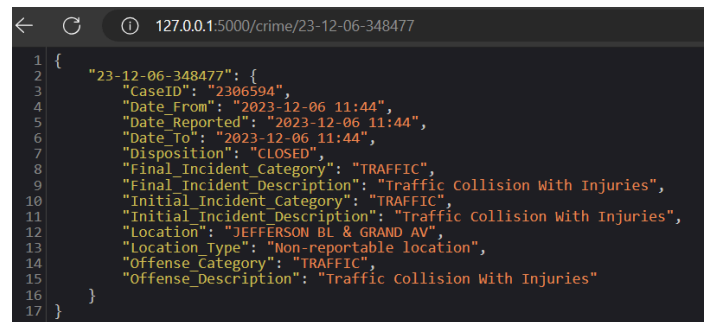
The project was developed with nine Flask app routes for calling functions in the database.py and rendering information from the HTMLs. The routes and their associated functions are detailed below.

/

This route renders home.html and performs no functions.

/crime/<id>

This route is hidden from the user and returns a JSON of the crime information associated with the entered id. This provides an alternative way to access the information.



```

1 {
2   "23-12-06-348477": {
3     "CaseID": "2306594",
4     "Date From": "2023-12-06 11:44",
5     "Date Reported": "2023-12-06 11:44",
6     "Date To": "2023-12-06 11:44",
7     "Disposition": "CLOSED",
8     "Final Incident Category": "TRAFFIC",
9     "Final Incident Description": "Traffic Collision With Injuries",
10    "Initial Incident Category": "TRAFFIC",
11    "Initial Incident Description": "Traffic Collision With Injuries",
12    "Location": "JEFFERSON BL & GRAND AV",
13    "Location Type": "Non-reportable location",
14    "Offense Category": "TRAFFIC",
15    "Offense Description": "Traffic collision With Injuries"
16  }
17 }

```

/search

This route renders search.html, with the results variable set to the default empty string. This ensures that results are only provided after a query. Categorical variables are also passed into the webpage for drop-down menus.

/results

This renders search.html along with the inputs to the search functions. Values entered into text boxes are extracted with the Flask request.args.get() function. Depending on which of the search functions is used, the entry variables are routed to their associated functions: search_case_id(), search_event(), or search(), the last of which being the most sophisticated.

If the search function returns a JSON object and success code 200, the length of matches is calculated to display on the search page, along with the retrieved information from the database. If the search function returns an error abbreviation and 7700, the corresponding error message will be displayed.

/reportacrime

This route renders reportacrime.html, which is the intermediate page requesting the password “PLEASE” to continue.

/reportacrimeadmin

This page retrieves the entered password from the intermediate page and renders reportacrimeadmin.html if the password is correct. This is where the inputs can be selected to create, update, or delete a report. Similar to the search page, categorical variables are also passed into the webpage for drop-down menus. All functions on this route pass variables to one of the following routes. However, if the password does not match, the user will be redirected to magicword.html.

/crime/inputs/<id>

This is the destination for the “Report A CRIME” submit button. If all fields are empty, it displays an error message “Please fill in at least one field.” It sends all inputs into the `report_case()` function. If successful, it renders the `crimes_submitted.html` with the automatically generated event number. If the function returns an error, it will display the appropriate error message on the same page.

/updateevent

This route is built similarly to the report case route. Successful update will render `crimes_updated.html` with a message confirming updates to the entry. Unsuccessful requests (case not found, invalid inputs, etc) will display the corresponding error message on the same page.

/deletecase

This renders the `crimes_deleted.html` with a confirmation message after receiving confirmation from the `delete_case()` function. Otherwise it displays the error message (“Case not found”, “Invalid input”, etc).

Learning Outcomes

In addition to the database concepts and query languages taught in lecture, this project was a great opportunity to see the impacts of hashing, scaling, distributed workflow, as well as the integration of Java and Javascript tools. For all team members this was the first time building a web application or working with databases, so the improvements in understanding have been substantial. An aggressive timetable for the project that focused on framing out the application functionally and then towards simplifying or streamlining operations allowed us to take time to assess our product along the way. Through multiple rehearsal demonstrations along the way we were able

to discover even the finest detail that could aid the user experience, such as alphabetically sorting the drop-down options.

Individual Contributions

Dan: Backend Design, Admin Interface, Data Manipulation Tools. Primary programmer for database.py.

Rodney: Database Management, User Interface, Data Preparation Tools

Kyle: Frontend Design, User Experience, Team Leader. Primary programmer for project.py and all HTML.

Conclusion

The application is user-friendly and can easily scale out for capacity by adding additional URLs to supplemental Firebase Databases if the user desires such capacity. The site provides all standard CRUD functions as well as an additional mass-entry function, specifically tailored to the data format published by the USC Department of Public Safety. It provides a useful reference tool for the Public Safety officials and an easy and insightful tool to inform the general user on USC Public Safety events.

Future Scope

Future efforts for this project could address security and user authentication as well as visualizations for the crime data. The information provided by the USC Public Safety office is not confidential or restricted in any way, but it would be very simple to link reports to files or repositories of supporting information, or digital workspaces focused on solving open cases. The contents of these investigations would be highly sensitive and should require very stringent security measures to provide unauthorized disclosure. These updates would benefit the administrative user, whereas visualizations

would benefit the greater USC community. Further work in this avenue would involve geotagging the reports or interpreting refined location data to a point where they could be used as input for a map overlay. This would allow users to physically see where these incidents are occurring, over time and by type, and generate even more useful insight from the databases.